Silence is Golden?

Adi Yoaz† Ronny Ronen† Robert S. Chappell‡ Yoav Almog†

†Intel Corporation ‡University of Michigan
{adi.yoaz, ronny.ronen, yoav.almog}@intel.com robc@eecs.umich.edu

A *silent store* is a write to memory that does not alter the value already present at the target address. Our experiments show that approximately 29% of dynamic store instructions are silent[1], with some applications showing upwards of 60% (gcc). At first glance, this may seem unintuitively large and suggestive of poor code writing and/or code compilation. However, we have found existence of silent stores to be more related to algorithm behavior and ISA translation than artifacts of code generation. Since silent stores can exist in significant numbers, the phenomenon is worth further investigation.

Molina *et al.* [2] and Lepak and Lipasti [1] both demonstrated that silent stores can be detected by the processor and then exploited to eliminate some unnecessary cache write-back traffic. However, these schemes required a preliminary load-and-compare operation to detect a silent store. This type of detection is somewhat wasteful, but more importantly, it limits the scope of practical silent store optimizations.

We propose speculative identification of silent stores to create more opportunities to exploit them. This paper introduces our silent store predictors and describes two ways they can improve performance: through enhancement of memory disambiguation and increased exploitation of dead instructions.

1 Silent Store Predictors

Accurate silent store predictors can be achieved by leveraging some existing branch prediction techniques. We predict a store instruction by accessing an array of counters by either the Instruction Pointer (IP) of the store or by a hash of the store IP and recent path information. The counter arrays are tagged by store IP, and each entry is either a 1-bit counter, a 2-bit counter[2], or a "sticky" bit that predicts either always silent or always non-silent. Table 1 shows a breakdown of accuracies for various predictor configurations.

As shown in Table 1, these predictors exhibit ability to predict silent stores accurately. The best configurations are over 90% accurate, while the chance of a harmful speculation (ps-ans) is kept to a minimum (see bold entries). All of the configurations in Table 1 have 32K entries and are 8-way associative. We have experimented with several predictor sizes and found accuracy to be fairly invariant for predictors from 8K entries to 128K entries. We chose 32K as the middle ground.

[1]Our experiments used X86 ISA traces of 3 benchmark suites (SPECint2000, sysmark98, and winstone98). Lepak and Lipasti measured 30–50% in similar experiments [1].

[2]Saturating counters are always initialized to "strongly not-silent".

Configuration	ps-as	ps-ans	pns-as	pns-ans	correct
IP 2-Bit	21.6%	3.8%	5.0%	69.5%	91.1%
IP 1-Bit	21.4%	5.3%	5.3%	68.1%	89.4%
IP Sticky-S	7.0%	**0.0%**	19.7%	73.3%	80.3%
IP Sticky-NS	6.5%	**0.1%**	20.2%	73.3%	79.8%
Path 2-Bit	19.0%	**0.9%**	7.7%	72.4%	91.4%
Path 1-Bit	21.2%	1.5%	5.5%	71.8%	93.0%
Path Sticky-S	17.2%	**0.3%**	9.4%	73.0%	90.3%
Path Sticky-NS	19.6%	14.6%	7.1%	58.8%	78.4%

Table 1: Silent store prediction accuracy breakdown.

ps-as	**Predicted silent, actually silent**	*success*
ps-ans	**Predicted silent, actually non-silent**	*causes recovery*
pns-as	**Predicted non-silent, actually silent**	*lost opportunity*
pns-ans	**Predicted non-silent, actually non-silent**	*success*

2 Enhancing Memory Disambiguation with Silent Stores

Memory disambiguation is the process of detecting memory dependence violations that occur due to speculative re-ordering of loads and stores. In [3], we proposed a dynamic memory dependence predictor to allow a load to bypass an earlier store if an address collision is unlikely. Our predictor used past collisions between pairs of loads and stores as the basis for prediction.

Memory disambiguation and dependence prediction can both be enhanced by exploiting silent stores. Clearly, a dependence violation between a load and an earlier *silent* store should not cause a recovery operation—the silent store is not providing any new data to the load. However, to gain the full benefit of this, our memory dependence predictor must identify silent stores early and allow loads to bypass them, *even if the load and store addresses are predicted to collide.* To achieve this, we propose including silent store prediction into our memory dependence predictor.

The potential performance of silent-store-aware disambiguation using a perfect silent store predictor is shown in Table 2. We see an average potential improvement of approximately 4%, with some individual benchmarks improving by almost 15%.

Our first attempt at realistically including silent store prediction into dependence prediction was very straightforward: we trained the dependence predictor only when a collision with a *non-silent* store was detected. The implementation change for this was also simple—to detect a silent store we compared the value

SPECint2000	bzip	crafty	eon	gap	gcc	gzip	link	mcf	perl	twolf	vortex	vpr	Aver
silent stores	17.32%	24.84%	41.76%	19.91%	61.21%	24.46%	27.71%	36.45%	47.17%	40.50%	39.46%	29.67%	34.20
perf. increase	1.3%	2.2%	7.7%	2.3%	11.2%	2%	1.6%	0.2%	5.4%	1.3%	1.4%	6%	3.55
sysmark98B	elast	excl	extrme	naturl	nscp	omnipg	ppnt	prdx	prem	pshop	word	xing	Aver
silent stores	2.04%	31.36%	22.72%	24.74%	21.40%	23.19%	26%	46.22%	24.69%	18.52%	38.77%	34.67%	26.27
perf. increase	8.5%	3.2%	14.7%	2.7%	2.8%	3%	1.8%	1.9%	8.4%	1.3%	3.6%	2.7%	4.55
winstone98B		acc	draw	excel	lotus	nav	pwrpnt	quat	tsksch1	tsksch2	wdpfct	word	Aver
silent stores		28.77%	24.96%	32.51%	25.57%	25.53%	27.53%	28.46%	33.74%	26.27%	23.60%	26.91%	27.62
perf. increase		1.6%	1.4%	1.6%	2.4%	2.1%	3.1%	3.7%	8.6%	6%	10.4%	1.7%	3.87

Table 2: Percentage of dynamic stores that is silent and potential performance (IPC) deltas due to inclusion of perfect silent store information into a memory dependence predictor. Machine configuration: 8-way superscalar issue, 512 micro-op window, 6 simple execution units, 2 complex units, 2 address generation units, 64KB I and D caches, 2MB unified L2 cache, 32K micro-op trace cache.

returned by the advanced speculative load with the value coming out of the store buffer. Note that we can take advantage of the speculative load to get the current memory value for the silent store comparison. Unfortunately, the performance gain using this simple change was far from the potential. Since the predictor was trained on only load-store collisions, it had trouble identifying silent stores that could be bypassed. This resulted in far fewer speculative load advancements than were possible.

Our second attempt integrated the "Path 2-Bit" predictor described in the previous section. This version was much better at identifying silent stores, and the proportion of the potential gained correlated well with the silent store prediction accuracy. We saw an average of 3% performance gain over the baseline.

3 Exploiting Dead Instructions and Silent Stores

Dead instructions are those which produce values that are never consumed by subsequent instructions. Most dead instructions result from unnecessary register saves and restores around procedure calls. Other dead instructions result from control flow structures in which register values are used on one path, but not on others. In both of these cases, it is difficult for the compiler to optimize away dead instructions without severely impacting code size.

Dead instructions can be squashed by the machine to reduce resource contention, thereby improving performance and reducing power. However, we have found that dead instructions account for only 1.7% of dynamic instructions (on average). This relatively small number of dead instructions limits the usefulness of such optimizations.

The silent store phenomenon presents an opportunity to effectively enlarge the set of dead instructions to exploit. As with "naturally" dead instructions, silent stores do not contribute useful work to the program and can be squashed to improve performance and save power. However, it is possible that the address and data computations of silent stores can also be squashed (if the computed results are not used by other useful instructions), further expanding the set. At this point, it is not clear how large the expanded set of dead instructions will be, though simply including silent stores yields an approximate total of 5% of the dynamic instructions.

If the set of dead instructions becomes large enough, it may become worthwhile to exploit. As mentioned above, "naturally" dead instructions can be squashed outright, giving obvious advantages. Theoretically, silent stores and their related computations could also be squashed. However, in reality, it is not possible to know if a store is silent until after the related computations have been completed. Therefore, we can only speculatively exploit silent store computations as "dead" by de-prioritizing them in various stages of the machine.

Many opportunities exist to exploit low priority instructions for performance or power. As clock speeds continue to increase, critical chip structures, such as the scheduling buffer, tend to shrink to meet timing constraints. Prior studies have examined prioritizing important chains of instructions to take best advantage of these limited resources [4]. Our methods for exploiting "dead" instructions fall along the same lines—we wish to *de*-prioritize *un*important chains of instructions, freeing bandwidth and reducing un-needed speculation. We are continuing research in this area, both to estimate the potential gains from such optimizations and to find new ways to exploit dead instructions.

4 Acknowledgments

We would like to thank Amir Roth (*amir@cs.wisc.edu*) and Evgeni Krimer (*evgeni.krimer@intel.com*) for their independent studies of our disambiguation mechanism and for their additional insights. We would also like to thank Anwar Rohillah for his work on modeling our prediction mechanisms.

References

[1] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[2] C. Molina, A. Gonzalez, and J. Tubella. Reducing memory traffic via redundant store instructions. In *Proceedings of the International Conference on High Performance Computing and Networking*, Apr. 1999.

[3] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[4] C. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.